6
Implementação

Este capítulo irá apresentar as tecnologias e os protocolos utilizados no Mobile Social Gateway e explicar de forma mais detalhada o papel de cada componente na arquitetura e o porquê de sua escolha.

6.1
Protocolo OAuth

As redes sociais utilizadas e estudadas neste trabalho possuem em comum a utilização do protocolo OAuth para autenticação de suas API's, conforme abordado no capítulo 3. O OAuth é um protocolo para acesso às API's de serviços web que permite ao usuário conceder às aplicações que ele utiliza o acesso a seus dados privados sem a necessidade de compartilhar sua senha. O protocolo utiliza pares de chaves públicas e privadas (*tokens*) para gerar assinaturas:

- um *token* de requisição garante que o usuário final autorize a aplicação;
- um *token* de acesso permite que a aplicação execute as operações da API.

A autenticação por meio do OAuth consiste de três passos:

1 - A aplicação cliente, representada neste trabalho pelo MoSoGw, solicita uma chave de autenticação à aplicação servidora (API do Twitter, Facebook etc), enviando a chave gerada no momento em que foi criada e um parâmetro contendo a URL para a qual o usuário será redirecionado após o processo de autorização. A resposta contém duas chaves, uma pública e uma privada:

- oauth_token: a credencial de identificação temporária;
- oauth_token_secret: a credencial secreta temporária;

2 - O usuário autoriza a aplicação cliente (MoSoGw) na rede social. Antes que o cliente solicite os *tokens*, o usuário será redirecionado para o servidor para que a requisição seja autorizada. O cliente constrói um URL de solicitação adicionando o parâmetro oauth_token, que contém a credencial de identificação temporária obtida na etapa 1. Depois de autorizar a aplicação, o usuário é redirecionado para a URL de callback.

3 - Após autorização do usuário, o MoSoGw troca a chave de autenticação pela chave de acesso, utilizada para acessar a API da rede social "em nome"do usuário. A requisição contém os parâmetros:

- oauth_consumer_key;
- oauth_token;
- oauth_verifier: contém o código de verificação recebido do servidor no passo anterior.

A resposta contém os seguintes parâmetros:

- oauth_token: o *token* identificador;
- oauth_token_secret: o *token* contendo a chave secreta.

Uma vez que a aplicação tenha recebido e armazenado os *tokens*, ela poderá realizar requisições autenticadas, que conterão os seguintes parâmetros:

- oauth_consumer_key: identificador do cliente;
- oauth_token: o valor do *token* usado para associar o pedido ao usuário;
- oauth_signature_method: o método usado para assinar a requisição. O protocolo OAuth fornece três métodos: HMAC-SHA1, RSA-SHA1 e PLAINTEXT;
- oauth_timestamp: o timestamp da requisição;
- oauth_nonce: string aleatória gerada pelo cliente para que o servidor verifique se o pedido já foi realizado;
- oauth_version (opcional): contém a versão do protocolo;
- oauth_signature: valor calculado pelo cliente.

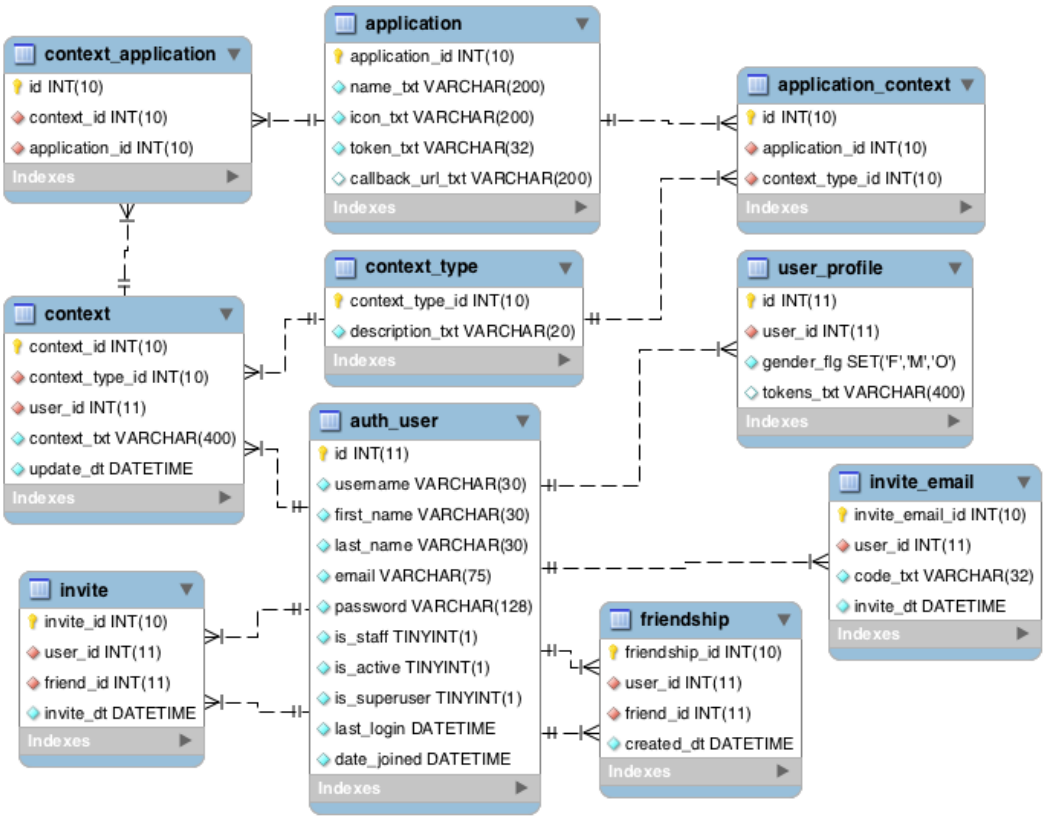

Figura 6.1: Diagrama Entidade Relacionamento

6.2 Diagrama de Entidade Relacionamento

O modelo de dados do MoSoGw é apresentando na figura 6.1. O modelo permite que as informações de contexto recebidas sejam armazenadas no banco de dados para serem consumidas de forma assíncrona por outras aplicações e web services de terceiros.

A entidade "context" armazena as informações de contexto enviadas para as aplicações, armazenadas na entidade "application". Novos tipos de contexto podem ser criados e armazenados na entidade "context_type". A entidade "context_application" associa a informação de contexto às aplicações selecionadas pelo usuário. A entidade "application_context" armazena as informações de assinatura de tipo contexto das aplicações de terceiros. As demais entidades permitem que sejam criados e gerenciados os relacionamentos entre os usuários do MoSoGw que formarão a rede social do *framework* cujo objetivo é apenas validar o filtro de sugestão de amigos (FriendsSuggestionFilter) sem a intenção de ser uma rede social complexa como são o Facebook e o Twitter.

O seguinte cenário exemplifica como os dados de contexto e a subscrição de tipos de contextos são mapeados pelas entidades descritas no parágrafo anterior: um usuário utiliza uma aplicação social para enviar informações de

sua posição ao Twitter através do MoSoGw. Por sua vez, um *web service* "CreateSocialActivityMap", desenvolvido por um terceiro, utiliza informações de contexto de localização para criar um mapa de densidade de usuários ativos do Twitter em uma determinada região e se cadastra no Mobile Social Gateway para receber essa informação. Nesse caso, o context_type será "posicao"e haverá uma entrada na tabela "application"com as informações associadas ao CreateSocialActivityMap e outra na tabela application_context, associando o tipo de contexto "posicao". Se o usuário da aplicação social enviar sua posição ao MoSoGw e permitir que o serviço CreateSocialActivityMap a consuma, será gerada uma entrada na tabela "context"contendo todas as informações associadas a esse contexto e outra entrada na tabela "context_application"relacionando o dado de contexto enviado (posição) e o serviço.

6.3 Tecnologias Utilizadas

O Mobile Social Gateway foi implementado em Python por acreditar que esta linguagem traria maior velocidade ao desenvolvimento. Além disso, cada componente de sua arquitetura de software foi escolhido para garantir que o *framework* possua alto desempenho e possa ser escalado para atender a milhares de requisições simultâneas. Quando se trata de escalabilidade de uma infraestrutura, existem duas abordagens. A escalabilidade vertical (*scale up*) significa adicionar mais recursos em um único nó do sistema (mais memória ou um disco rígido mais rápido) para atender a demanda crescente de processamento e armazenamento. A escalabilidade horizontal (*scale out*) significa adicionar mais nós ao sistema permitindo a distribuição do trabalho entre múltiplas máquinas.

Nesse contexto, a arquitetura modular do MoSoGw tem uma característica importante: é possível prever a quantidade de máquinas necessárias para lidar com a quantidade desejada de carga e assim realizar a escalabilidade horizontal da infraestrutura. Em outras palavras, se cada *box* responde, por exemplo, a 250 solicitações por segundo (req/s) e a carga desejada é de 2500 req/s, então serão necessárias 10 máquinas. Neste caso, seria necessário incluir na arquitetura um servidor balanceador de cargas responsável por distribuir os acessos entre cada uma dessas instâncias do MoSoGw de forma a equalizar o consumo de recursos de *hardware*.

Ainda sobre escalabilidade, o problema C10k (8) refere-se à dificuldade que os servidores web possuem para lidar com um grande número de clientes ao mesmo tempo (C10k significa dez mil conexões simultâneas). Uma série

de fatores devem ser considerados para permitir que um servidor web suporte muitos clientes e envolvem uma combinação de restrições do sistema operacional e limitações do servidor web mas basicamente ela ocorre devido à forma com que os servidores trabalham com cada conexão recebida. Embora existam alguns servidores web que podem lidar com mais de 10 mil conexões, esta é a limitação de conexões simultâneas da maioria dos servidores de aplicação.

A arquitetura orientada a eventos, conhecida como *asynchronous non-blocking I/O*, foi concebida em resposta ao C10k. A principal vantagem da abordagem assíncrona é a escalabilidade. Em um servidor orientado a processo (como o Apache), cada conexão simultânea é tratada por uma *thread*, o que gera um *overhead* significativo no sistema operacional. Um servidor assíncrono, por outro lado, é orientado a eventos e trata as solicitações com uma única (ou muito poucas) *threads*.

A tecnologia escolhida deve consumir poucos recursos da máquina, contribuindo para escalar verticalmente a arquitetura. Além disso, a utilização de uma infraestrutura de cache em diversos pontos da arquitetura reduz a necessidade de processamento e de I/O no banco de dados.

O nginx (9) e o Tornado (33) foram escolhidos por possuirem uma arquitetura orientada a eventos e não bloqueante. O nginx é conhecido por ser um servidor HTTP bastante rápido e por utilizar poucos recursos da máquina. O Tornado, por sua vez, é um framework Python bastante enxuto e de alta performance. Conforme será explicado, as requisições ao framework chegam pelo nginx, que repassa as requisições para uma das instâncias do Tornado. Nesse ponto, é importante ter os dados cacheados e evitar acesso ao banco de dados. O sistema de cache escolhido foi o memcached.

A comunicação com as redes sociais pode gerar um gargalo na aplicação pois nas situações e que elas estão recebendo um grande volume de acessos, o seu tempo de resposta pode ser alto. Assim, essa comunicação deve ser assíncrona e, para tal, utilizou-se uma aplicação gerenciadora de filas chamada "Beanstalkd". Dessa forma, quando um contexto chega, ao invés dele ser enviado para a rede social, ele vai para a fila do Beanstalkd. Paralelamente, existe um processo (daemon) que roda de tempos em tempos no servidor e consome essa fila. Ele lê um item empilhado e, então, envia o dado de contexto para a rede social correspondente.

Nesta seção, serão detalhadas as principais tecnologias e componentes utilizados na implementação da arquitetura proposta e como cada uma dessas escolhas contribui para atingir os objetivos propostos e garantindo a escalabilidade do *framework*.

6.3.1 Tornado

O principal componente do MoSoGw é o Tornado, um servidor web escrito em Python e desenvolvido pelo Facebook. O Tornado é distinto da maioria dos servidores web tradicionais por ser non-blocking e consideravelmente mais rápido e leve, apresentando baixo consumo de recursos. Por ser *non-blocking* e usar *epoll* ou *kqueue*, ele pode lidar com milhares de conexões simultâneas, o que também é uma característica importante para lidar com o problema C10K.

Uma característica importante do Tornado e que o torna mais leve é possuir um baixo acoplamento entre os módulos Python com os quais foi desenvolvido. Dessa forma, o desenvolvedor poderá decidir quais aspectos do servidor tornado ele gostaria de utilizar e será o responsável pelo desenvolvimento do *core* da aplicação web.

O maior obstáculo ao desenvolver uma aplicação web é o correto dimensionamento dos processos para atender a aplicação. Como a aplicação Tornado é executada como um servidor HTTP *stand-alone* diretamente acoplado às classes da aplicação, é necessária a execução de vários processos para servir múltiplas requisições. Cada um desses processos estará disponível em uma porta do sistema operacional. Assim, é necessária a utilização de um servidor web para gerenciar os múltiplos processos.

6.3.2 Nginx

O MoSoGw utiliza o nginx, servidor HTTP conhecido por ser muito rápido e usar poucos recursos de CPU e memória. Ao contrário dos servidores tradicionais, o nginx não usa *threads* para fazer conexões e sim uma arquitetura assíncrona orientada a eventos.

O nginx possui uma característica importante que influenciou na sua escolha como servidor HTTP: ele realiza o balanceamento de carga entre cada um dos processos da aplicação que executam em portas diferentes e, se um dos processos parar de executar, automaticamente o nginx deixará de encaminhar novas requisições durante um certo período de tempo até que o processo volte a executar.

6.3.3 SQLAlchemy

Em relação à conexão ao servidor de banco de dados, a arquitetura proposta utiliza o SQLAlchemy (34). O SQLAlchemy é uma biblioteca de

mapeamento objeto-relacional (ORM: Object-Relational Mapping) SQL em código aberto desenvolvido para a linguagem de programação Python.

O SQLAlchemy permite:

- o gerenciamento de sessão por *transactions*, *rollback*;
- a escrita de *queries* mais complexas e próximas ao SQL sem forçar o usuário a utilizar a ORM;
- a construção de *queries* complexas a partir de trechos de código evitando que o usuário tenha que escrevê-las explicitamente;
- a troca do banco de dados sem necessidade de rescrita das *queries*, ou seja, o usuário pode trocar o MySQL pelo Oracle de forma simples.

6.3.4 Beanstalkd

O Beanstalkd (35) é um servidor simples e rápido para o enfileiramento de mensagens, organizadas em "tubos". A aplicação cliente pode inserir e consumir mensagens de e para tais tubos. A interface do beanstalkd é genérica mas foi originalmente concebida para reduzir a latência de visualizações de páginas em aplicações web de alto volume de acessos, executando tarefas demoradas de forma assíncrona. Uma vez que o Beanstalkd não possui mecanismos de autorização / autenticação, ele deve ser executado em uma rede confiável.

As vantagens do Beanstalkd são:

- simplicidade: seu protocolo é baseado no protocolo do memcached;
- totalmente especializado em fila de jobs;
- rapidez por utilizar filas na memória, sem I/O extra no disco rígido do servidor;
- facilidade na configuração;
- bibliotecas disponíveis para várias linguagens de programação que abstraem a comunicação com o servidor.

6.3.5 Memcached

O memcached (36) é um sistema de *caching* distribuído de objetos em memória baseado em chave-valor concebido para ser utilizado em qualquer sistema mas principalmente para aumentar a performance e escalabilidade de aplicações web. Possui código fonte aberto e um conjunto de APIs para várias linguagens de programação que facilitam sua utilização. Atualmente, o memcached é usado por alguns dos mais acessados sites da Internet como o Facebook, Youtube e Flickr.

Ele é frequentemente usado para aumentar a velocidade de processamento de sites dinâmicos orientados a banco de dados através do *caching* de dados e objetos em RAM para reduzir a quantidade de acessos ao banco de dados ou a uma API.

O memcached foi escolhido por possuir uma integração bem simples com o servidor nginx: quando determinada URL é requisitada e existe no memcached, o nginx obtém diretamente o dado armazenado sem repassar a requisição para a aplicação.

6.4 Arquitetura de Software

A arquitetura do Mobile Social Gateway (figura 6.2) possui um design modular adequado para executar em um ambiente de nuvem (*cloud*) ou *cluster* (portanto, escalabilidade vertical): uma instância de nginx distribui a carga entre várias instâncias do Tornado (setas 1). O balanceamento de carga é totalmente gerenciado pelo nginx e garante que cada instância de Tornado trabalhe com a mesma carga das outras. O número de instâncias depende das configurações do servidor e é proporcional ao número de processadores.

Além desses dois componentes, o MoSoGw se conecta a um banco de dados MySQL (seta 2) para armazenar os contextos, os usuários e as informações de amizade, convites enviados e algumas outras informações. Essas informações são armazenadas em memória utilizando o memcached (seta 3). O conjunto de várias instâncias do Tornado e uma de nginx é definido como *box*. Conjuntos de *boxes* podem ser espalhados em qualquer nó do cluster (será melhor explicado no capítulo 9).

A integração com as redes sociais pode ser computacionalmente cara e relativamente lenta devido à necessidade de implementação do protocolo OAuth e de algum tipo de pré-processamento que possa ser feito com as informações enviadas ou recebidas. Assim, para alcançar um maior desempenho, a distribuição de informações de contexto para as redes sociais foi dividida em duas

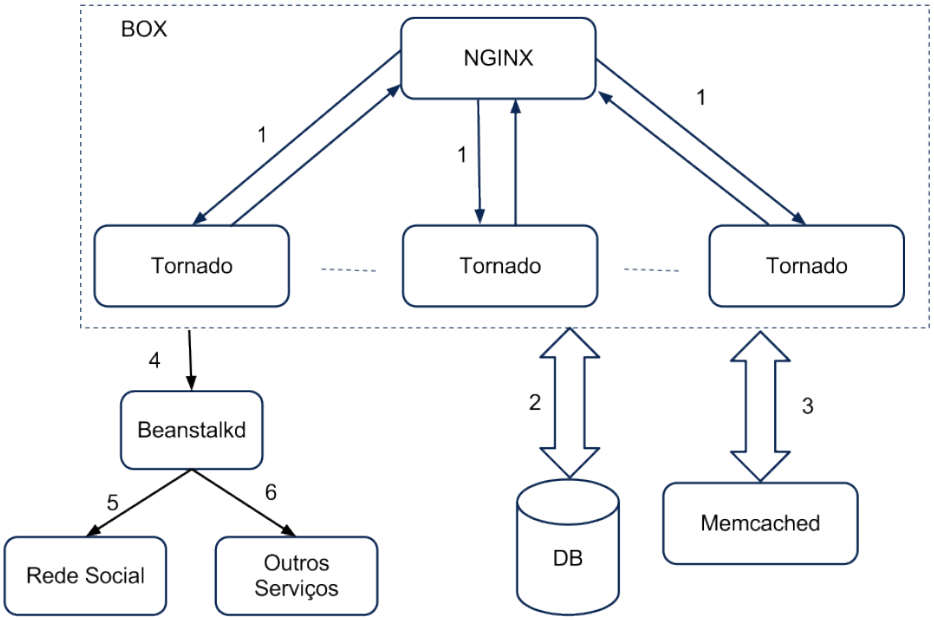

Figura 6.2: Mobile Social Gateway Box

etapas: primeiro, os dados de contexto enviados por um cliente móvel são enfileirado no Beanstalkd (seta 4) com algumas informações adicionais (por exemplo, para qual rede social os dados devem ser enviados).

Em segundo lugar, um processo isolado consome os dados desta fila e, para cada entrada, envia os dados de contexto para a rede social (seta 5) desejada ou para um outro serviço (seta 6). Este padrão produtor-consumidor pode levar a situações complexas de competição (condições de corrida) mas, felizmente, o Beanstalkd trata de forma eficiente todas estas questões. Assim, o consumidor da fila do Beanstalkd será a única interface com as redes sociais e com os web services de terceiros e para que seja feita uma integração com uma nova rede social basta alterar um único ponto de código, conforme melhor explicado na seção 6.8.

Se observarmos mais de perto essa arquitetura (figura 6.3) e examinarmos apenas a aplicação Python, ela será composta por uma instância de Tornado que receberá a requisição encaminhada pelo nginx e a enviará para o handler específico, que pode buscar um dado no banco de dados ou renderizar um *template*. As classes de repositório procuram a informação desejada no cache e se não for encontrada, a procuram no banco de dados. A resposta pode ser um JSON simples ou algo mais complexo, como um arquivo html. Neste último caso, o handler renderiza um arquivo de *template* passando as informações necessárias para montar o arquivo html. O *template* é simplesmente um arquivo html com algumas marcações específicas e que será pré-processado (junto com todos os dados necessários) por um gerenciador de *templates*, como

o Mako (37). O *template* pode conter trechos de código, *loops* do tipo *for* e diversas outras funcionalidades dependendo do gerenciador de *template*. Em Java, por exemplo, os *templates* são representados pelas JSPs.

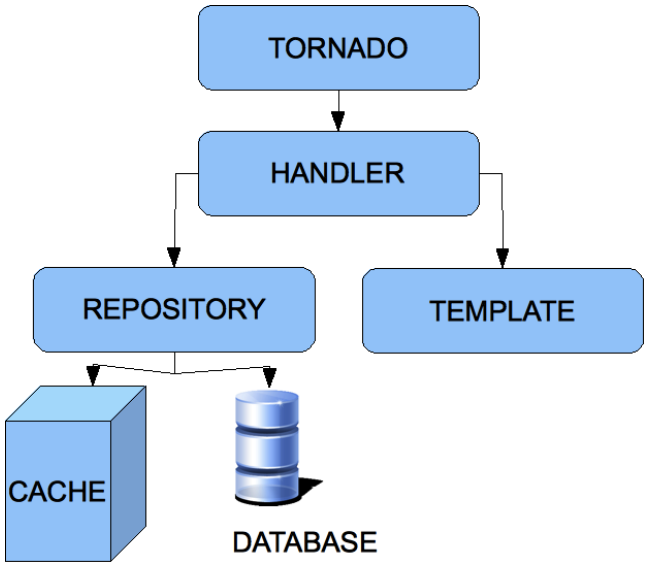

Figura 6.3: Mobile Social Gateway

6.5 Camadas de Cache

O processo de escrita e leitura (I/O) pode ser bastante custoso para o banco de dados em situações de muitas conexões simultâneas e criar um gargalo na aplicação, comprometendo o desempenho (a aplicação passa a atender menos usuários) e até mesmo a disponibilidade (a demora na resposta do banco de dados pode enfileirar conexões e travar a aplicação). Para evitar o reprocessamento desnecessário das requisições já realizadas, foram criados três níveis de *caching* no MoSoGw: cache em disco (nginx), cache em memória (memcached) e um cache de sessão do SQLAlchemy.

6.5.1 Nginx

Como primeira camada de cache, o nginx pode ser configurado para salvar em disco a resposta a uma determinada solicitação. Pode-se definir diferentes áreas de cache, cada uma com uma configuração específica de tempo de validade e tamanho máximo. O nginx irá gerar um *hash* com a URL requisitada e a resposta será salva no diretório definido em proxy_cache_path. O nginx possui uma heurística para criar os diretórios de cache, criando subdiretórios de acordo com a primeira letra do *hash* criado. Então, na próxima vez que um mesmo pedido for feito, esse cache será usado. A fim de garantir uma entrada de

cache específica para cada usuário, seu ID é usado como entrada no algoritmo de geração da chave de cache utilizado pelo nginx (nginx proxy_cache_key).

6.5.2 Memcached

Ao contrário do nginx, o memcached usa a memória e é a segunda camada da arquitetura de cache, responsável por armazenar as respostas das solicitações ao banco de dados. Ele obtém o par (<key>, <value>) e armazena na memória. A regra de formação da chave é bastante simples: ela é composta pelo nome do módulo Python (incluindo o pacote), pelo nome da classe e pelo nome do método com os parâmetros. Existe uma restrição no tamanho da chave de memcached, que deve possuir apenas 32 caracteres. Assim, a partir da pré-chave é gerado um *hash* MD5 contendo 32 caracteres. Exemplo:

```
Pacote: mss.models
Módulo: user
Classe: User
Método: ids()

Chave: mss.models.user.User().ids()
Chave MD5: 76952f477043e7bfdfd33e37eda943da
```

Para reduzir o espaço de memória utilizado, em vez de armazenar o resultado de uma solicitação ao banco de dados, armazena-se a lista de IDs retornados, e para cada ID, o objeto retornado (que pode ser usado em pedidos de outros controladores) também é armazenado.

O cenário a seguir ilustra essa arquitetura: quando um usuário solicita uma lista de todos os amigos, o MoSoGw guarda no memcached a lista de IDs encontrados (e somente IDs). Para cada ID, o objeto correspondente também será completamente salvo na memória. Assim, se fizermos um pedido para o handler que retorne um amigo específico, por exemplo, este objeto já estará na memória. Em outras palavras, quando a consulta dos ids é executada, a aplicação verifica se essa chave está no cache. Se não estiver, a *query* é executada e a resposta é cacheada. Para cada ID retornado, a consulta para obter o objeto correspondente será executada. Novamente, o sistema procura pela chave correspondente no cache. Se encontrada, o valor é retornado. Caso contrário, a *query* será executada e o objeto será cacheado. Dessa forma, a listagem dos ids será invalidada no cache apenas se um usuário for incluído ou removido.

Este processo também aumenta o desempenho do sistema pois se uma lista de objetos (não IDs) fosse armazenada, quando um objeto fosse alterado

deveria-se invalidar a lista inteira. Da maneira proposta, o será necessário expirar apenas o objeto (o ID nunca vai mudar) e esta atualização irá refletir em todos os serviços que compartilham o objeto.

6.5.3 SQLAlchemy

O SQLAlchemy realiza o cache em sua sessão dos objetos das classes Python instanciados. No momento em que um objeto é instanciado pela aplicação ele também é criado na sessão do SQLAlchemy. Na medida em que a aplicação conclui as operações envolvendo esse objeto (alteração de uma propriedade, por exemplo), ele é inserido no banco de dados. Então, quando for novamente solicitado, ele será encontrado na sessão. O SQLAlchemy implementa um *flag* de validade do objeto, ou seja, se alguma requisição alterar o objeto, ele vai ter um *flag* indicando que algo mudou e será inserido no banco de dados e atualizado na sessão.

6.6 Interfaces de Programação

Os recursos existentes no MoSoGw estão disponíveis através de duas APIs. Em ambas as APIs, antes de efetuar qualquer operação, a aplicação deverá seguir o fluxo básico de autenticação do usuário no MoSoGw. Essa autenticação independe das redes sociais e é importante para garantir que o usuário utilizando o MoSoGw é quem ele diz ser. Dessa forma, a aplicação não poderá enviar dados de contexto para os perfis de outros usuários que não aquele autenticado. O fluxo de autenticação do usuário da aplicação social é composto por 3 passos:

1. O usuário cria uma conta no MoSoGw, fornecendo nome e email;

2. Em caso de sucesso, a senha gerada será enviada para o email do usuário, como forma de verificar o endereço. O usuário será convidado a alterar essa senha;

3. De posse da senha, o usuário poderá, então, se autenticar no MoSoGw. Ao fazê-lo, um *token* será gerado e deverá ser utilizado pela aplicação social nas próximas requisições como forma de autenticação.

O *token* gerado será utilizado como uma chave de cache que armazenará o perfil do usuário. A cada requisição, a aplicação verificará se o *token* é válido, ou seja, se a chave existe no memcached, e, em caso afirmativo, carregará o perfil do usuário. De posse desse perfil, o MoSoGw pode identificar quais

informações estão associadas ao usuário que as solicitou garantindo que apenas ele terá acesso as suas informações.

O *token* obtido no primeiro serviço deverá ser enviado como parâmetro auth em todas as requisições subsequentes. O retorno (formato JSON) sempre conterá um código HTTP e uma mensagem de confirmação da ação que será omitida na tabela 6.1.

6.6.1 API REST

Essa API é composta por diversos serviços REST, cujas respostas estão no formato JSON (39). Neste caso, os aplicativos devem implementar as solicitações HTTP e conhecer as interfaces e URLs do MoSoGw. As URIs desses serviços e uma breve descrição de seu funcionamento podem ser vistos na tabela 6.1.

6.6.2 API JAVA

A segunda API é escrita em Java para a plataforma Android sendo a melhor indicação para desenvolvedores de aplicativos móveis nesta plataforma. Uma vez que os detalhes de HTTP e JSON são abstraídos pela API, os programadores só necessitam instanciar uma classe e, a partir dela, obter acesso a todos os recursos do MoSoGw. Esta API é usada pelos aplicativos protótipos descritos na seção 7.4 e o conjunto de funcionalidades implementadas pode ser visto na tabela 6.2.

Em ambos os casos, cada vez que um nova rede social e web services são adicionados ao MoSoGw pelo desenvolvedor do *framework*, eles são reconhecido pela API. Assim, os desenvolvedores das aplicações sociais precisam apenas ser conscientes deste dinamismo e preparar sua aplicação para isso. Dessa forma, é interessante que seja utilizado o serviço de descoberta de novas redes sociais provido pelo MoSoGw. Através desse serviço, pode-se elaborar a lógica de exibição das redes sociais para o usuário final da aplicação móvel.

6.6.3 API com Demais Serviços Processadores de Contexto

Conforme descrito nas seções anteriores, o MoSoGw possui uma interface que expõe os dados de contexto armazenados e permite que novos tipos de dados de contexto sejam criados. Dessa forma, outros serviços cadastrados no MoSoGw, podem utilizar essas informações para construir outras mais complexas ou utilizar de alguma forma diferente.

Um serviço externo poderia processar as informações de posição enviadas por diversos usuários através de aplicações sociais como o Mobile Social Share (7.4.1 e, então, criar um mapa de usuários do MoSoGw ativos por região. Outro uso mais interessante é criar um mapa das condições do tráfego por região, ou seja, através da posição de *posts* seguidos de vários usuários em uma mesma região e do cálculo da velocidade, pode-se estimar a condição do tráfego nesta região. Essa informação mais complexa ("Trânsito na Gávea") pode ser enviada ao MoSoGw e compartilhada nas redes sociais.

O MoSoGw também provê uma interface para criação de novos tipos de contexto, permitindo que os web services definam seus próprios contextos, se necessário. Um exemplo da necessidade de criação de novos contextos seria uma aplicação social que processe a informação de velocidade no próprio dispositivo do usuário. Assim, ao invés de passar a posição do usuário para que o servidor a processe, ela envia o contexto de velocidade. Esse novo tipo de contexto, que não existe por padrão no framework, poderia ser criado através da API do MoSoGw. Nesse caso, seria necessario que a aplicação social seguisse os passos de registro descritos a seguir e, então, enviasse o nome e a descrição desse novo contexto através da API Rest ou Java correspondente.

O processo de registro de novos serviços externos no MoSoGw é relativamente simples e consiste em um HTTP POST para o framework contendo os tipos de contexto em que eles estão interessados e uma URL a ser chamada quando novos dados de contexto de seu interesse chegarem (url de *callback*). Após o registro, o novo serviço recebe um ID único que o identifica e que deverá ser passado a cada requisição feita ao MoSoGw. Concluído o processo, o serviço estará disponível para o desenvolvedor da aplicação social, que poderá então decidir se deseja disponibilizá-lo para o usuário final. Se optar pela exibição, e o usuário permitir que seus dados de contexto sejam enviados para o serviço (através de um *check box* da GUI, por exemplo), esses dados serão armazenados na base de dados do MoSoGw e estarão disponíveis para serem consumidos pelo serviço. Quando um novo dado de contexto for recebido, todos os serviços de terceiros que assinaram esse tipo de contexto receberão um POST HTTP através da URL de *callback* informada. Alternativamente, os aplicativos podem ativamente obter dados de contexto, solicitando-os de forma síncrona através da API do MoSoGw.

6.7 Interface de Administração

O processo de autorização nas redes sociais descrito na seção 6.6.3 necessita de um *token* específico para cada usuário que deve ser obtido pela

aplicação social móvel e repassado ao MoSoGw. No entanto, essa complexidade foi removida da aplicação e transferida para o *framework* através de uma interface de administração na qual o usuário da aplicação móvel se autentica com seu login da rede MoSoGw e passa a ter acesso às suas informações de cadastro (perfil) e pode se autenticar nas redes sociais seguindo o fluxo *client* para obtenção do *token* OAuth.

O sistema de administração também prevê um usuário com privilégios de administrador e que possui acesso às aplicações cadastradas, aos perfis dos usuários e às demais entidades do MoSoGw. Esse usuário é criado no momento em que o desenvolvedor da aplicação móvel configura esse sistema no servidor em que sua instância do MoSoGw será executada. Esse acesso permite que usuários sejam bloqueados, alterados ou mesmo removidos em caso de uso indevido. Da mesma forma, serviços de terceiros podem ser excluídos do sistema e bloqueados.

6.8 Extensibilidade

Para facilitar o processo de integração com as diferentes redes sociais, a arquitetura do Mobile Social Gateway provê uma classe de interface que deve ser implementada por cada uma das classes desenvolvidas para lidar com a integração. Essa interface possui métodos de autenticação e envio de dados para as redes sociais. Caso seja necessário a integração com o MySpace, por exemplo, basta a criação de uma classe que implemente os métodos da interface descrita. Essa classe tratará as questões de conexão e autenticação de forma genérica de tal forma que o MoSoGw tenha apenas que chamar a classe genérica.

Nesse sentido, o Beanstalkd (seção 6.3.4) possui um papel importante nessa integração por ser o ponto único onde se dá a integração entre o *framework* e as redes sociais ou web services de terceiros e é nele que a nova classe deve ser adicionada.

A extensibilidade também está na possibilidade de se adicionar filtros para as informações provenientes das redes sociais com a mesma facilidade e da mesma forma com que novas redes são adicionadas através da implementação de interfaces. O processo de filtragem e processamento das informações de contexto pode, inclusive, ser feito por outras aplicações que subscrevam dados de contexto do MoSoGw permitindo, assim, a adição de novas funcionalidades sem a necessidade de alteração do código do *framework*.

Tabela 6.1: API Rest

Módulo	Descrição	URL	Parâmetros	Retorno
App /app	retorna redes sociais e web services de terceiros	/get.json	-	lista de aplicações
	cadastra uma aplicação	/create	nome, ícone, url de retorno	token
	remove uma aplicação	/remove	*token*	-
	assinatura de contextos	/subscribe	*token* e lista de contextos	-
	remove uma assinatura	/unsubscribe	*token*	-
	obtém os contextos disponíveis	/contexts.json	*token*	-
Usuário /user	autentica um usuário	/login	-	*token* (auth) de autenticação do usuário
	criar um usuário	/create	username firstName lastName gender	email com a senha gerada
	obter as informações disponíveis de um usuário	/user.json	username (opcional)	usuário desejado
	buscar usuários	/search.json	username buscado	usuários encontrados
	regerar a senha	/login/rescue	username	email com a nova senha
Amigos /friends	obter os relacionamentos	/get.json	-	com todos os amigos encontrados
	remover um relacionamento	/remove	username	-
	sugerir novos relacionamentos	/suggestions	-	lista de amigos sugeridos
Convite /invite	enviar um convite para um usuário cadastrado	/send	username	-
	aceitar um determinado convite enviado por sistema	/accept	id do convite	-
	obter os convites enviados por um usuário	/get.json	-	convites encontrados
	enviar um convite por email para um usuário não cadastrado	/email/send	email	-
	aceitar um determinado convite por email	/email/accept	id do convite	-
	obter convites pendentes de um usuário	/invites.json	-	-
Contexto	enviar contexto	/context	-	-

Tabela 6.2: API Java

Método	Descrição
void Initiate()	Verifica o *status* dos serviços
String Login(String username, String password)	Realiza a autenticação do usuário e retorna o *token*
String CreateUser(String lastName, String firstName, String username, String gender)	Cria um novo usuário
String SendContext(Map<Integer, String> context, String auth)	Envia dados de contexto para as redes sociais e serviços especificados
String GetUserInformation(String username, String auth)	Obtém as informações de um usuário
String Search(String username, String auth)	Procura por usuários
String AcceptInvite(String username, String auth)	Aceita um convite para amizade
String SendInvite(String username, String auth)	Envia um convite para amizade
String RemoveFriendShip(String username, String auth)	Remove uma amizade
String GetFriend(String auth)	Obtém informações sobre um amigo
String GetInvitations(String auth)	Obtém a lista de convites pendentes recebidos
String SendEmailInvite(String email, String auth)	Envia um convite por email para um usuário se registrar na aplicação
String GetFriendsSuggestions(String auth)	Obtém as sugestões de amizades